A METHOD AND SYSTEM FOR DETERMINING TOTAL CODE EXECUTION TIME IN A DATA PROCESSOR

TECHNICAL FIELD

The invention refers to a method and a processing system for a communications network, according to the non-characterizing portions of claim 1 and 8, respectively.

BACKGROUND

In processor technology, such as data packet processing technology, more specifically in the entering of instructions for the processes, traditional linker algorithms uses large chunks of code, i.e. machine code chunks from an assembler. Traditional linkers also have an object file where the code chunk is stored along with relocation objects. The linkers place many chunks of code sequentially in a memory and link the chunks of code together using of the relocation objects. The codes are optimized by memory utilization, i.e. all codes are placed in sequence.

A disadvantage with known processor assembly-linking algorithms is that it is difficult to meet processing requirements of the processor during programming and compiling. More specifically, it is difficult to include real time requirements of the data processing, when programming and compiling using traditional linker algorithms.

SUMMARY

It is an object of the present invention to present a method and a processing system for a communications network, at which it is easier to meet processing requirements of data processes in the network, during implementation of instructions for the processes.

It is also an object of the present invention to present a method and a processing system for a communications network, at which it is easier to take under consideration real time requirements of the data processing in the network, during programming and compiling of instructions for the processes.

The objects are achieved by a method and a processing system for a communications network, according to the characterizing portions of claim 1 and 8, respectively.

Dividing the program code into a plurality of sequences, defining, based on the program code, a plurality of relocation objects, each corresponding to a dependency relationship between two or more of the sequences, and allocating the sequences to a processor instruction memory, provides a structure of the codes that make them easy to manipulate in order to meet data processing requirement of the communications network.

Preferably, at least one directed graph is formed, based on at least some of the sequences and at least some of the relocation objects, and a longest execution path through the directed graph is determined. Sequences in the instruction memory can be moved and state preserving operations can be entered, so as to make at least two execution paths equally long. This provides an effective tool for controlling the code in order to meet real time requirements in the communication network. More specifically, the invention facilitates, as opposed to known processor assembly-linking algorithms, often designed to optimize memory utilization, the determination of the total execution time for each alternative execution path, avoiding difficulties in meeting processing time requirements.

BRIEF DESCRIPTION OF FIGURES

Below, the invention will be described in detail, with reference to the accompanying drawings, in which

- Fig. 1 shows schematically the structure of an instruction memory, according to a preferred embodiment of the invention, in relation to programmed instructions and a data path,
- Fig. 2 shows schematically the instruction memory in Fig. 1 and a part of it enlarged,
- Fig. 3 shows schematically the structure of a program according to a preferred embodiment of the invention, and
- Figs. 4-7 show examples of program structures,

DETAILED DESCRIPTION

Here, reference is made to Fig. 1. The processing system according to the invention is adapted to process physical data passing through a processing pipeline 1. The data can be in the form of data packets. The direction of the data stream is indicated by an arrow A. The processing system can be a PISC (Packet Instruction Set Computer), as described in the patent application SE0100221-1. The processing pipeline 1 comprises a plurality of pipeline elements 2. Between the pipeline elements 2 engine access points 3 are located, at which, for example, data packet classification can take place.

The processing pipeline 1 can be adapted to a so called classify-action model. Thereby, in the processing of a data packet, first a classification is performed, e.g. a CAM lookup. A matching classification rule can start an action with an argument. The action can be a process program executed in a pipeline element 2 and that performs a task as a directed graph of sequences starting with a root sequence, as described below. The processing system can be adapted to run many forwarding plane applications simultaneously, e.g. both MPLS and IPv4. The applications can be separated logically, for example by special tags for classification and different sets of actions (packet programs).

The processing system, comprises an instruction memory 4 for storing instructions for the data processing. The instruction memory 4 comprises rows 5 and columns 6. Each pipeline element 2 comprises a number of instruction steps for the data process, whereby each instruction step is allocated to a column 6 in the instruction memory 4. Accordingly, each pipeline element 2 is allocated a certain number of columns as indicated by the broken lines B. The rows 5 each corresponds to an address in the memory 4.

In a method according to a preferred embodiment of the invention, described closer below, an assembler generates sequences 7 of instruction words 8, preferably machine code instruction words, e.g. VLIW instruction words, and a linker place the sequences into the instruction memory 4 and link the sequences together. Referring to Fig. 2, in the memory 4, each sequence occupies memory space, in the same row and in adjacent

columns. Each sequence is a row of machine code instruction words that will be executed consecutive. After executing an instruction word the processor will normally execute another instruction word in the next column at the same row.

As depicted in Fig. 1, the processing system comprises a process program, which corresponds to a directed graph 9 of sequences 7 that each perform a certain task on data, e.g. data packets, such as forwarding.

As depicted in Fig. 3, the directed graph starts with a root sequence 7a. It is limited to one pipeline element 2. Each root sequence 7a is a sequence at the beginning of a process program, and can be marked as "Root" in the program. A root sequence 7a can be started as a result of a CAM classification or an instruction in the form of a jump in a sequence in a preceding pipeline element 2. The linker can export a start row of the root sequence so that a run-time software can map action to classification rules.

The directed graph ends in leaf sequences 7b. Each leaf sequence 7b is a sequence that ends with a relocation instruction that the packet program should exit or jump to a sequence in another pipeline element 2. Thus, the linker can link directed graphs, or programs, in different pipeline elements 2, by connecting, or linking, a leaf sequence in one program to a root sequence in another program.

The directed graph 9 comprises branches 10. Each branch 10 is a relocation object that provides information that there is an alternative sequence to jump to at the instruction at which the branch is located. By using a branch 10 the processor can perform a jump to another sequence at another row, but a default branch is executed at the same row and belongs to the same sequence.

A sequence exit can be any of the following: A relocation object instructing a jump to another sequence in the same pipeline element 2, a relocation object instructing a jump to

a sequence in a following pipeline element 2, or an instruction to end the program. Any of the two latter alternatives form an exit in a leaf sequence 7b.

The relocation objects result in the process program having a finite number of alternative paths until the program exits in one or many exit points (leafs).

In a method for a communications network according to a preferred embodiment of the invention, the assembler receives a program code, comprising a plurality of instructions for the communications network. The assembler divides the program code into a plurality of sequences 7, e.g. sequences of a PISC code, and defines, based on the program code, a plurality of relocation objects 10, each corresponding to a dependency relationship between two or more of the sequences 7. The assembler forms at least one directed graph 9, based on at least some of the sequences 7 and at least some of the relocation objects 10, the directed graph having one or many roots.

In other words the assembler or "compiler" divides the code into "atomic" tasks, activities or sequences that have dependencies to each other such that they need to be performed in a consequence order.

The directed graph 9 is stored as an object file. The object file comprises a code sequence format and a relocation object format that define the dependencies between the sequences. Each code sequence has a length (a number of instructions).

The directed graph is analyzed. The linker validates that the directed graph consists of one to many partial ordered sets. This means determining the existence of any circle reference by any of the relocation objects 10 between any of the sequences 7.

Preferably, the method comprises the linker validating that the code meet a predetermined time requirement, i.e. hard execution time requirement. This is done with a longest path algorithm over the directed graph, i.e. determining a longest execution path through the

directed graph, which includes adding sequence lengths or lengths of partial sequences. In the pipeline processing case the execution time requirement is limited by the number processing stages. In a more general case the time limit can be any hard real-time requirement.

The linker places the sequences in the instruction memory, as described above. The two-dimensional instruction memory 4 allows many alternative execution paths to be stored in parallel. Hence, the program having to be stored as a directed graph.

Referring to Fig. 4, the linker moves at least one sequence in the instruction memory and allocates at least one state preserving operation, or no operation instruction, NOP in the instruction memory, so as to make at least two execution path equally long, whereby the length of the at least two execution paths correspond to the longest execution path. As an example, in Fig. 4 the longest execution is determined by the root sequence 7a, a first relocation object 101 and a first leaf sequence 71. An alternative execution path is formed by a part of the root sequence 7a, a second relocation object 102 and a second leaf sequence 72. The alternative execution path has a shorter execution time compared to the longest execution path, due to the second relocation object 102 being located closer to the root of the root sequence 7a than the first relocation object 101, and the second leaf sequence 72 being shorter than the first leaf sequence. The linker moves the second leaf sequence and enters state preserving operations NOP before and after the second leaf sequence 72. Alternatively, the second leaf sequence is not moved and state preserving operations NOP are entered after second leaf sequence 72. Thereby, all alternative execution paths become equally long, and the execution time is equal to the longest path for all possible alternative paths.

Referring to Fig. 5, in an alternative embodiment, a special no operation row 12 in the memory 4 is used by the linker, whereby leaf sequences 7b in execution paths being shorter than the longest execution path, can jump by means of a relocation object 10, to

the no operation row 12 when finished. It is important that said jump is carried out to the correct position in the no operation row 12 so that execution paths become equally long.

Fig. 6 depicts a situation where alternative execution paths are to be synchronized to a shared sequence 7c. The linker moves a sequence 73 in a path being shorter than the longest execution path, and enters state preserving operations NOP before and after the moved sequence 73.

Fig. 7 depicts an alternative to the method described with reference to Fig. 6. The linked can add a no operation sequence 14 on the same row as, but before the shared sequence 7c. A relocation object 103 is entered to make the shorter execution path jump to the no operation sequence 14 before entering the shared sequence 7c. This provides for an easier programming of the processor, since any execution path being shorter than the longest one, can be extended to correspond the latter simply by entering a relocation object at its end pointing to the defined no operation sequence 14, or, referring to Fig. 5, the no operation row 12.

The invention guarantees that the packet program maintains its state to the next engine application point (EAP) even after the packet program has exit. The linker can use a super root with the length of zero instruction as the root for all packet programs, which makes the graph operations more easy.

As has been described the invention provides for the sequences to be moved and the state preserving operations to be entered in such way that sequences that are dependent on each other are synchronized.

Above, the invention has been described pointing out its usefulness for a program code for a pipelined processing system, such as a PISC processor. However, the invention is suited for program codes for any hard real-time system, e.g. on a traditional processor, where the exact execution time is critical.